MIT/LCS/TR-362

SIMULATING APPLICATIVE ARCHITECTURES ON THE CONNECTION MACHINE

Bradley Clair Kuszmaul

June 1986

This blank page was inserted to preserve pagination.

Simulating Applicative Architectures on the Connection Machine

by

Bradley Clair Kuszmaul

S.B. Computer Science and Engineering, M.I.T. (1984)

S.B. Mathematics, M.I.T. (1984)

Submitted to the

Department of Electrical Engineering and Computer Science

in partial fulfillment of the

requirements of the degree of

Master of Science

in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

June, 1986

Signature of Author _____

Department of Electrical Engineering and Computer Science

May 9, 1986

Certified by _____

Jack B. Dennis

Thesis Supervisor

Accepted by _____

Arthur C. Smith

Chairman, Committee on Graduate Students

Simulating Applicative Architectures on the Connection Machine

by

Bradley Clair Kuszmaul

S.B. Computer Science and Engineering, M.I.T. (1984)

S.B. Mathematics, M.I.T. (1984)

Submitted to the Department of Electrical Engineering and Computer Science on May 9, 1986 in partial fulfillment of the requirements of the degree of Master of Science in Electrical Engineering and Computer Science at the Massachusetts Institute of Technology

Abstract: The connection machine (CM) is a highly parallel single instruction multiple data (SIMD) computer, which has been described as 'a huge piece of hardware looking for a programming methodology'[Arv]. Applicative languages, on the other hand, can be described as a programming methodology looking for a parallel computing engine. By simulating architectures that support applicative languages ('applicative architectures') (e.g., data flow and reduction architectures) on the CM we can achieve the following goals:

- Quickly and easily experiment with the design and implementation of applicative architectures.
- Run large applicative programs effeciently enough to gain useful experience.
- Support programming environments that allow us to do general purpose computation on the CM.

We describe the techniques which we use to simulate applicative architectures on the CM, and the discuss implications for the generalized case of simulating multiple instruction multiple data (MIMD) systems on single instruction multiple data (SIMD) computers. We describe the results of our simulations, concluding that the CM can run applicative programs efficiently, even though the CM was not explicitly designed for that task.

Thesis Supervisor: Jack B. Dennis

Title: Professor of Computer Science and Engineering.

Keywords: Applicative Architectures, Connection Machine, Combinators, Data flow, Simulation.

To Kimberly, who made me take walks on the Riverwalk when I thought I should be working on my thesis. It is because of her that I am still a living, breathing human.

Contents

List of Figures

Acknowledgements

I would like to thank my thesis advisor, Jack Dennis, for his guidance and encouragement, Mike Dertouzos for finding money for me when no one else could, and Guy Steele for helping to deal with the hair of doing off campus research.

Thinking Machines Corporation provided a connection machine[1] and software support. Too many TMC people are involved to name them all, but they include Danny Hillis, Cliff Lasser, Brewster Kahle, John Rose and Guy Steele.

[1]The phrase "connection machine" is a registered trademark of Thinking Machines Corporation.

Chapter 1

Introduction

The connection machine (CM) is a highly parallel single instruction multiple data (SIMD) computer[Hil85], which has been described as 'a huge piece of hardware looking for a programming methodology'[Arv], while applicative architectures can be described as a programming methodology for parallel computation looking for a parallel computing engine.

We have simulated architectures which support applicative languages ('applicative architectures') on the CM in order to try to understand how to design scalable architectures to perform efficient general purpose parallel computing.

We define a *general purpose parallel computer* to be a computer that can exploit parallelism, when present, in any algorithm[AI85]. A *scalable architecture* is an architecture that allows us to add hardware resources, resulting in higher performance, without requiring substantial rewriting of application programs[AI85]. By *efficiency*, we mean that we would like the performance of the architecture to improve linearly (if possible) with the amount of hardware resources and the amount of parallelism apparent in the algorithm.

Applicative architectures, such as data flow and combinator reduction architectures, are efficient, general purpose, scalable architectures, and much work has been done to demonstrate this[AI85]. We are interested in specific implementation issues, and have built efficient

parallel simulators for several applicative architectures on the CM. In this context, efficient means that as we increase the size of the connection machine, we should be able to get a corresponding linear improvement in the performance of the simulators (once again assuming sufficient parallelism in the algorithms being run on the simulated architectures).

This simulation allows us to quickly and easily experiment with new ideas for applicative architectures, dramatically reducing the expense of such experimentation, in much the same way as MIT's proposed Multiple Processor Emulation Facility[ADI83] will when completed.

We have designed simulators for three applicative architectures:

- Static data flow[Den74],
- VIM style dynamic data flow[Den80], and
- Combinator reduction[Tur79].

We have actually implemented the static data flow and combinator reduction simulators. In this paper we describe the design of those simulators for running on the CM, and some lessons we have learned from the design and implementation of these simulators.

1.1 Background

The connection machine is an SIMD machine, in which every processor has some local state and a connection to a router network. The connection machine processing elements are *bit serial*, which means that it requires quite a few clock cycles to perform a complicated operation such as floating point multiply. There is a front end computer (sometimes called a *host*) which interacts with the connection machine. In order to reduce the required bandwidth between the host and the CM processors, there is a microcontroller, which receives high level instructions (such as 'add two numbers') from the host, and issues many low level bit operations to the connection machine processors.

The applicative architectures that we are considering are all multiple instruction multiple data (MIMD) architectures; each processor may need to do something different with its data.

In order to simulate such architectures, we need to be able to simulate an MIMD machine on the CM. We will exploit the fact that there are not really very many different kinds of things that can happen in an applicative architecture (e.g. in a static data flow architecture, there are only a few different kinds of graph nodes, and in a combinator reduction architecture, there are only a few different combinators). Our general strategy is to cycle through each of the different things that could be done by a processor. Given a thing to be done, there is some state transition of the processors which want to do that thing. We select those processors which 'want' to do that thing, and issue the SIMD instructions to perform that state transition.

This paper assumes that the reader is familiar with the applicative architectures that we are simulating.

1.2 Results

We have learned a few important lessons for running complex programs on the connection machine: Writing programs directly in low level connection machine languages is very difficult, and it helps to have structured programming languages to keep this complexity under control. It is also important to carefully choose low level primitives; in particular, the choice of message passing primitives is very important.

We can run relatively large applicative programs efficiently enough to gain useful experience in the design of programming constructs and program development tools for applicative programs.

In some sense, our simulators are an existence proof of the statement that the connection machine can do general purpose parallel computation: We can run functional languages on the connection machine.

1.3 Floor plan

Chapter 2 provides an abstract description of the connection machine by describing some 'high level instructions' for the connection machine. These high level instructions are called *macro instructions*. Since the connection machine microcontroller can be programmed, the set of macro instructions can be chosen to suit the needs of the programming community. We will describe how some of the more complex instructions that we have found useful are implemented.

Chapter 3 describes the applicative architectures that we are simulating, along with the design of our simulator for each of those architectures. We note that the choice of macro instructions can dramatically effect the design and complexity of the simulators.

In order to control the complexity of writing MIMD programs for a SIMD machine, we designed a language called StarTalk which allows us to simulate MIMD architectures on a connection machine. Chapter 4 discusses the motivation for designing a new programming language for the connection machine, and describes the StarTalk language and our StarTalk compiler, including the optimizations which are performed. The simulators were written in StarTalk.

Chapter 5 concludes with some preliminary performance measurements and a discussion of our conclusions and some remaining open problems.

Chapter 2

The Connection Machine

The connection machine (CM) is a highly parallel single instruction multiple data (SIMD) computer[Hil85]. This section provides an abstract description of the connection machine, along with enough of a description of the hardware to make it possible to understand how to code the algorithms that we will describe later in this paper.

2.1 Connection Machine Hardware

The prototype connection machine under construction at Thinking Machines Corporation, Cambridge, MA, consists of the following:

- There are 64K processing elements (PE's).
- There are 4K bits of local memory associated with each PE.
- There are a few one bit flags in each PE.
- Each PE is a one bit ALU.
- Each PE is connected to a *router* which delivers messages between PE's through a network with a hypercube topology.

- There is also a two dimensional communication network, which we we will ignore in the rest of this paper.
- There is a microcontroller which receives high level commands (macro instructions) such as 'add two fields' from the front end computer and issues bit operations to the PE's.
- There is a front end computer which serves as the user interface to the connection machine. The front end computer issues macro instructions and provides a program development environment for the user.

2.2 Connection Machine Software

One of the fundamental concepts for programming the CM is the 'currently selected set' (CSS) of PE's. Most macro instructions are executed only by the PE's in the CSS, so that, e.g. during an ADD macro instruction, only those PE's in the CSS will actually perform the ADD, while the other PE's do nothing. We will describe some macro instructions, with the caveat that these macro instructions are not be the macro instruction set implemented for the actual CM microcontroller: The exact macro instruction set supported on the CM differs in being richer than the instruction set we describe here, and in certain other ways, such as the choice of names and arguments of operations. A data item always consists of a string of bits having consecutive addresses within the memory of a PE. Such a bit string is called a *field*, and can be characterized by its starting address in the 4K local address space, and a length. A *pointer* is a field containing enough bits to specify the absolute address of a PE (i.e. the PE number). The CSS is stored in one of the flags mentioned above (e.g. the *context-flag* in PE number i contains a one if and only if PE number i is in the CSS).

2.2.1 Context Manipulation Macro Instructions

There are three context manipulation functions which, together, can perform all context manipulation operations that we might desire. The context manipulation operations are very

fast, and each runs in only a few clock cycles.

`cm:set-context`

Sets the CSS to be all the PE's:

$$\text{CSS} \leftarrow \text{The set of all PE's.}$$

`cm:load-context` *m*

Takes *m* as a memory address. Let $M(m)$ be the set of PE's for which memory location *m* contains a one. (Note that, as always, "memory locations" are addresses in the processors' local memories.)

$$\text{CSS} \leftarrow \text{CSS} \cap M(m).$$

I.e. the context is conditionally loaded from memory location *m*.

`cm:store-context` *address*

In all processors currently selected, a one is stored at location *address*, and at all the processors not currently selected, a zero is stored at location *address*.

It is possible to perform any context manipulation with these primitives, but for performance reasons one might be interested in defining other context manipulation functions.

2.2.2 Arithmetic Macro Instructions

We will describe one of the 'arithmetic' macro instructions in detail, and hint at the richness of the instruction set that is actually available. These instructions involve no inter-PE communication, or PE to front-end-computer communication. They change the local state of the PE's (i.e. the flags and the local memory).

`cm:lognot` *destination source length*

The *destination*, *source*, and *length* values are integers. The *destination* and *source* are addresses in the 4K local memory of the PE's, and the *length* is the number of bits on which to

operate (i.e. the field length). Each bit of the *destination* field is set to the the logical not of the corresponding bit of the *source* field. This operation takes a few clock cycles for each bit of the operation.

Similarly, there are operations to perform the bitwise logical and operations, and other logical operations. Since the connection machine is microcoded, it is possible to define these operations as two address or three address instructions. Near the other end of the complexity spectrum we find floating point operations, such as three address floating point multiply, which might take three addresses as an input and perform floating point multiplication conforming the IEEE floating point standard. The more complex operations can take longer: We might expect a floating point operation to take a thousand clock cycles.

2.2.3 *Global Or Wire* Operations

There is a *global or wire* which allows the host machine to find out if any processor satisfies a given predicate. The *global or wire* is actually implemented as a tree of *or* gates (see Figure 2.1). The macro instruction that uses this wire is as follows:

`cm:global-or` *location*

Returns a zero if and only if all the selected processors have a zero in location *location*, otherwise returns a one. This is a very fast instruction, and takes only a few clock cycles.

2.2.4 Message Sending Macro Instructions

There are several ways of sending data from one processor to another. We will ignore the two dimensional grid in this paper, and concentrate on the ways to communicate over the general purpose routing network.

`cm:send-data` *dest-memory dest-proc source-memory size notify-bit*

Sends data from PE's in the CSS to other PE's. The PE that is to receive a message is

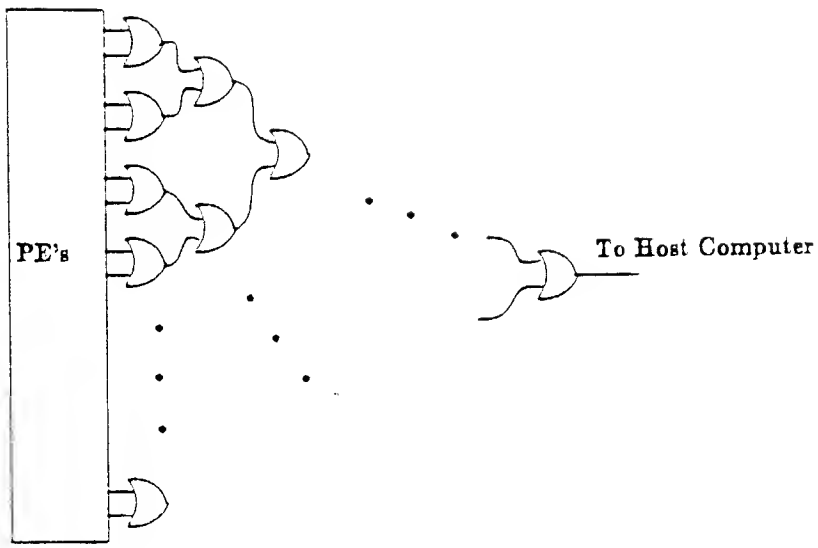

Figure 2.1: The *global or wire* is implemented as a global or tree.

specified with its absolute address starting in location *dest-proc*, which contains a *pointer*. Note that this allows the destination address to be computed locally by each PE. The data being sent starts at location *source-memory* and is of length *size*. The data is deposited at the field of length *size* starting at location *dest-memory* in those PE's that receive messages (the *dest-memory* field is unchanged in other PE's). Memory location *notify-bit* is set to 1 in those PE's that receive messages, and 0 in those that do not, independently of whether they are in the CSS. If more than one message is received at any given PE, that PE receives a message, but the data deposited in *dest-memory* is unspecified (it may be one or the other of the messages, or it may be totally garbled). This specification allows that if m messages arrive at a given processor, the time for completion of the `cm:send-data` instruction may be at least linear in m (i.e. the time can be $\Omega(m)$) due to serialization of the arriving messages without violating this specification).

The simple collision handling mechanism defined by `cm:send-data` places important constraints on the design of applications that use routing.

- If two processors need to send data to a third processor, and the data must not be

garbled, the software must arrange at a higher level that the two processors do not send the data in the same **cm:send-data** instruction.

- Even if the data can be garbled (an example of this would be where the information being sent is a boolean, and the fact that a message has arrived at all can be construed as a **true**), the **cm:send-data** instruction still does not do the job if a large number of processors need to send the value to a single destination (it is too slow). The problem here is due to the serialization done by **cm:send-data** at the destination.

- There is no way for a processor to efficiently retrieve data from another processor (without resorting to high level conventions). In particular, if processor A contains data, and processors $B_1, B_2, \ldots, B_m$ need to access that data, it may take linear (in m) time to get the data distributed to the B's. If, on the other hand, every m is small, this sort of fetch operation can be done by having each processor sends its own address to the processor being fetched from, and that processor will send its data to the processor whose address it has just received. An implementation for such a fetch operation is shown in Figure 2.2.

The original solution that we proposed to these problems was to use a higher level software convention to work around the problems in **cm:send-data**. One design involved copying data whenever too many pointers to the processor containing the data are needed, and another design uses *fan-in trees*.

Fan-in trees work as follows (See Figure 2.3): If m processors need to communicate with one, then a tree of processors, of depth $\Omega(\log m)$ is built, and all communication is done through the tree. In particular, if processors $B_1, \ldots, B_m$ need to send a value to processor A, then the B's send it into the tree, and the internal tree nodes, when they receive one or two messages, forward one message toward the root. In this way, processor A is guaranteed not to receive too many messages, and the messages will not be garbled. The tree can also be used to retrieve data from processor A, by sending a request for the data towards the root of the tree, and letting the data from A be sent back towards the leaves of the tree. Also, any

```
(defun cm:get (from-proc dest source length)
  ;; The WITH-SCRATCH-SPACE form allocates temporary memory in
  ;;  all the processors.  E.g. the lisp variable TEMP-CONTEXT is
  ;;  bound to an integer which is the address of one bit of scratch
  ;;  memory, and GETER-ADDRESS is bound to an integer which is the
  ;;  address of a field which can contain a pointer to another
  ;;  processor.
  (with-scratch-space ((temp-context 1)
                       (geter-address *address-size*)
                       (remaining-context 1)
                       (received-p 1))
    (cm:store-context temp-context)
    (cm:store-context remaining-context)
    (while (not (zerop (cm:global-or remaining-context)))
       ;; Each processor sends its own address.  Note that every
       ;;  processor stores its own address in a location specified
       ;;  by *SELF-POINTER*.
       (cm:send-data geter-address from-proc
                     *self-address* *address-size* received-p)
       ;; now the guys who received a message should reply
       (cm:set-context)
       (cm:load-context received-p)
       (cm:send-data dest geter-address source length received-p)
       ;; now the guys who received the reply drop out of the
       ;; computation
       (cm:set-context)
       ;; set remaining-context to zero if received-p
       (cm:lognot received-p received-p 1)
       (cm:logand remaining-context received-p 1)
       ;; and load the context to try again
       (cm:load-context remaining-context))
    ;; now restore the original context
    (cm:set-context)
    (cm:load-context temp-context)))
```

Figure 2.2: A naive implementation of `cm:get` which works if not too many processors are fetching from any given processor.

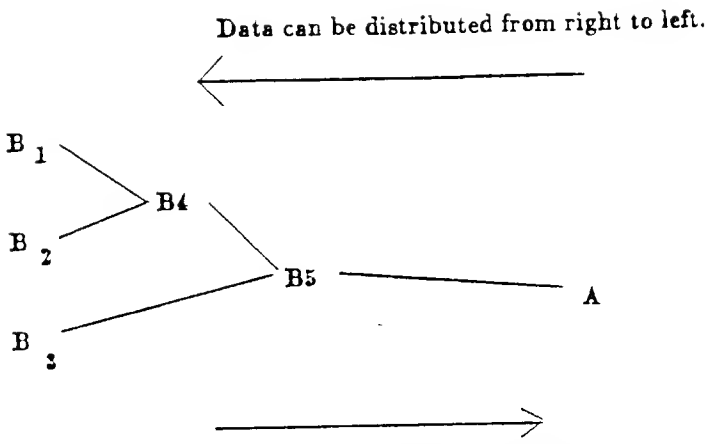

Figure 2.3: Fan-in trees are a high level convention which can handle the problems associated with using cm:send-data.

associo-commutative operation can be used to combine messages when such a tree has been built (e.g. the contents of the messages can be added together, or they can be bitwise or'd).

We actually implemented such fan-in trees for our static data flow simulator, and they were responsible for most of the complexity of the interpreter, as we shall show in Section 3.1, where we discuss the implications of using fan-in trees or other high level software conventions to avoid the problems inherent in cm:send-data. The complexity of implementing these fan-in trees is almost overwhelming in the simulation of combinator graph reduction and VIM style dynamic data flow. Thus, we define several operations to address these issues. We will give the specifications of the operations, along with a sketch of their implementations on the connection machine.

cm:send-with-add *dest-memory dest-proc source-memory size notify-bit*

Sends data as for cm:send-data, except that if any two messages are being sent to the same location, their data is added together. In order to achieve good performance, this addition must, in general, occur before the messages actually arrive at their destination (e.g. the

addition could happen as soon as two messages collide in the network).

`cm:send-with-logior` *dest-memory dest-proc source-memory size notify-bit*

Sends data similarly to `cm:send-with-add`, except that when messages collide, their data bits are bitwise inclusive or'd together.

We also define an instruction for fetching data from other processors:

`cm:get` *from-proc dest source length*

Each selected processor ends up with its memory field at location *dest* of length *length* containing the data that was in the field at location *source* of length *length* in the processor named in the pointer field at *from-proc*.

Figure 2.2 contains an implementation of `cm:get` which is correct except that it is too slow.

All of these more general operations must be implemented by using fan-in trees at some level, either in hardware or in software.

The NYU Ultracomputer[DGK86] implements these fan in trees in hardware: The trees are built on the fly by the routing network. The connection machine could use the same tricks as the Ultracomputer, except that it would have to simulate the tricks in software (and they would be implemented in microcode). We use another technique (described below) to implement these abstract routing operations.

Since the prototype connection machine does not support general routing operations such as `cm:send-with-add` and `cm:get` directly in hardware, a software simulation is done. The algorithm used is to sort the messages, using a standard parallel sorting algorithm[BRR,Kus85], according to the destination address (See Figure 2.4). After sorting the messages, we have the property that messages going to any given PE are in adjacent PE's. The PE's can then, in one step, form themselves into linked lists (each list containing messages going to one particular PE). Given the linked lists, the PE's can form themselves into balanced binary trees by first

passing their own addresses one element to the left (forming links which span two PE's), and then two elements to the left (forming links which span four PE's) and so on. The combining can then be done in $\log n$ phases, where n is the maximum number of messages going to any processor.

It is also possible to use the sorting and scanning trick mentioned for `cm:send-with-add` and shown in Figure 2.4 to do the `cm:get` operation quickly.

We can generalize these operations a little further, and design an operation that builds processors into a tree.

Definition 1 *A class of binary trees is* roughly balanced *if the maximum depth of any tree is a polynomial in the log of the number of vertices in that tree.*

`cm:build-tree` *key key-length root-p left left-p right right-p*

The `cm:build-tree` operation builds trees out of processors. All the processors with the same key are formed into a tree. The key is stored in a field of length *key-length* at memory address *key*. Thus if there are n distinct keys actually appearing in various selected processors, we will end up with n trees. The trees are binary trees, which are required to be roughly balanced. (I.e. there is a polylog function which says how unbalanced the tree can be.) Each processor has the single bit at memory location *root-p* set to one if it is the root of the tree in which it appears, otherwise it has the one bit field at *root-p* set to zero. Each processor that has a left child, has the one bit field at *left-p* set to one and the pointer field at *left* set to the address of that left child. If there is no left child, then *left-p* is set to zero and the data in the pointer field at *left* is undefined. Similarly, *right-p* is set to one iff the processor has a right child, and if so, *right* contains the address of that child.

Our early designs explicitly maintained the fan in trees needed to implement these abstract operations. Our final design uses microcoded versions of these operations to build fan in trees on the fly. One can argue that by repeatedly building the fan in trees over and over, we

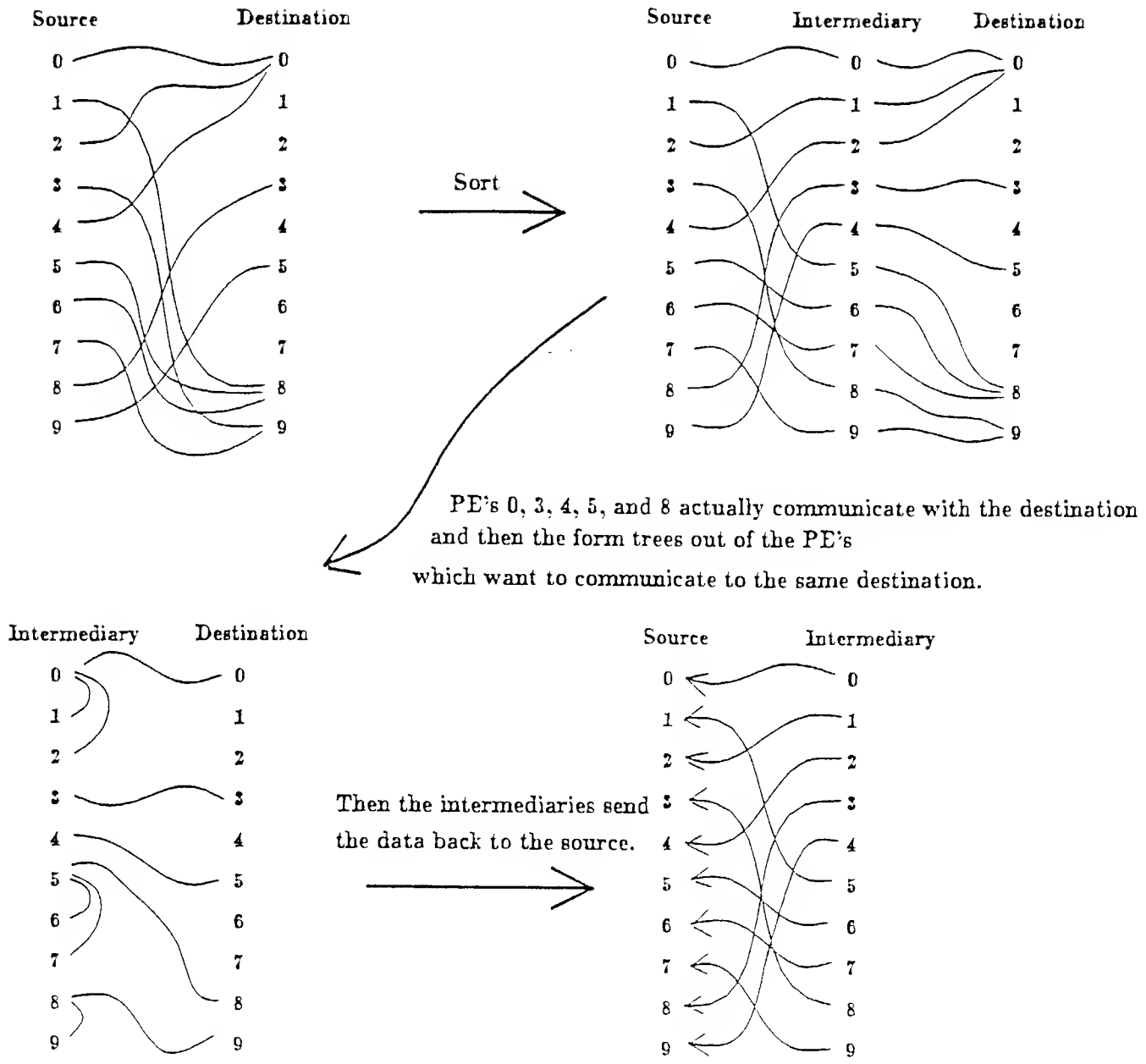

Figure 2.4: Sorting and scanning to make `cm:send-with-add` run quickly.

have wasted computating resources, but we believe that the large constant factor gained by microcoding these routines offsets the expense of rebuilding the fan in trees every time they are needed.

2.2.5 Consing Macro Instructions

By 'consing' we mean the operation of allocating resources. In our case, we are interested in allocating PE's. It is easy to allocate PE's one at a time under the control of the front end computer, but we are interested in allocating PE's in parallel. In particular, we would like the following operation.

`cm:cons` *new-address want free*

Here, *new-address* is the address of a *pointer*, *want* is a memory address which specifies which PE's want to cons (i.e. those with memory location *want* set to one), and *free* is a memory address which specifies the set of PE that are free. This operation is done regardless of the CSS. Let W be the set of PE's for which *want* is one, and F be the set of PE's for which *free* is one (it is not necessary that W and F be disjoint, although it is expected that in most applications F and W will be disjoint). If $|F| < |W|$ then the result is unspecified. Otherwise, each element of F is given the address of a unique PE in W, and in each PE from F that is allocated, *free* is set to zero.

This `cm:cons` operator gives us much power. For example, we can copy a graph in constant time (actually, in time which is proportional to the maximum degree of the graph), once the graph has been identified (see Figure 2.5). To do this we cause each element of the graph to cons up a free PE, then forward the address of that free PE to its neighbors in the graph, and then, having received addresses from its neighbors, sending those addresses to the new PE[Chr84].

The `cm:cons` operation is of sufficient complexity that we believe that an explanation of its implementation is in order. The `cm:cons` operation is implemented in terms of an 'enumerate'

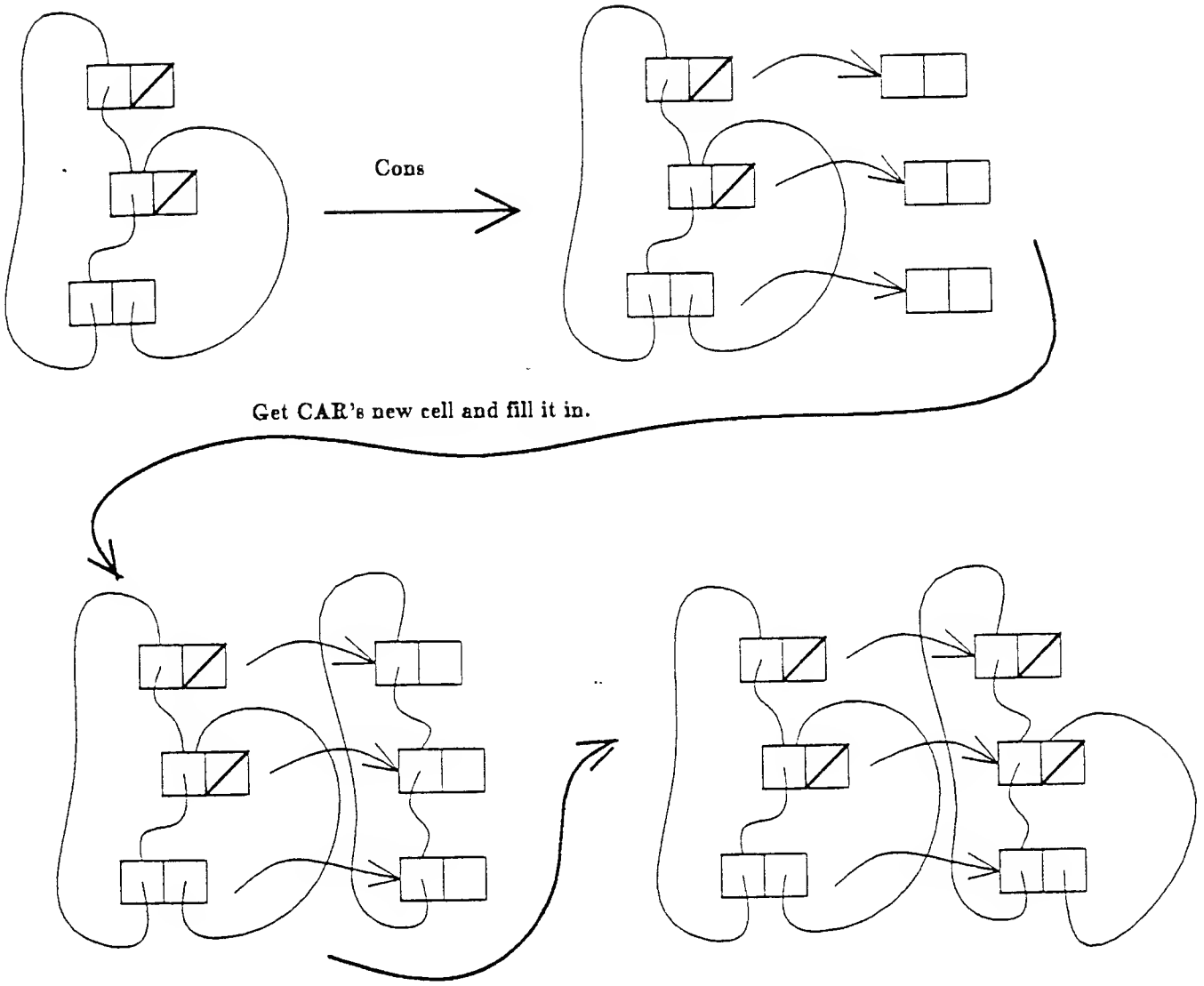

Figure 2.5: Copying graphs in constant time.

operation:

cm:enumerate *number*

The *number* argument is the address of a *pointer*. Let n be the size of the CSS. For each PE in CSS the *pointer* starting at *number* is set to a unique integer in the range 0 (inclusive) to n (exclusive). The value n is returned as a result of the call to **cm:enumerate** in the front end computer.

If we are given **cm:enumerate**, we can implement **cm:cons** by performing an enumeration of the free set, and an enumeration of the set that wants to cons. Let n be the minimum of the size of the set that wants to cons and the size of the free set. Each PE sends its own address to the value it received during the enumeration. For each $0 \leq i < n$ there is one PE f in the free set and one PE w in the consing set which received i during their respective enumerations, thus PE i will receive two messages, one containing f and one containing w. Then PE i sends f to w, and now w has the address of a unique free PE. A program which implements **cm:cons** in terms of **cm:enumerate** appears in Figure 2.6

In order to implement **cm:enumerate**, we can use subcube induction, which is a specialization of tree induction. In subcube induction, we statically organize all of the PE's into a balanced binary tree with PE's at the leaves (see Figure 2.7). We then enumerate inductively on the size of the tree. A tree containing one PE gets a one if that PE is selected, or a zero otherwise. A tree of depth k first enumerates its subtrees, and determines the number of PE's in each of the subtrees. If there are l PE's in the left subtree and r PE's in the right subtree, then each of the PE's in the right subtree is instructed to add l to its value, and we know that there are $l + r$ PE's in the tree of depth k.

On a complete butterfly network (with processors at all the internal nodes) the running time is $O(\log N)$. It is possible to simulate a complete butterfly on the CM for the enumerate operation with an extra $O(\log \log N)$ time[BRR] (where N is the number of processors in the CM). The **cm:enumerate** macro instruction has been "speed hacked" to run faster than a

```
(defun cm:cons (new-address want free)
   ;; get some scratch memory
   (with-scratch-space ((temp-context 1) ;; temp-context is one bit
                        ;; allocate some space for enumeration
                        (want-addr *address-size*)
                        (free-addr *address-size*)
                        (conser    *address-size*)
                        (consee    *address-size*)
                        (received-conser 1)
                        (received-consee 1)
                        (ignore-bit 1))
      ;; now save the context, so we will be able to restore it.
      (cm:store-context temp-context)
      (let ((number-wanting nil)
            (number-free nil))
        ;; unconditionally load the context with WANT, and enumerate them
        (cm:set-context)
        (cm:load-context want)
        (setq number-wanting (cm:enumerate want-addr))
        ;; and send ones own self-address to the rendezvous point
        (cm:send-data conser want-addr *self-address* *address-size*
                      received-conser)
        ;;
        ;; now do the same for FREE
        (cm:set-context)
        (cm:load-context free)
        (setq number-free (cm:enumerate free-addr))
        (if (< number-free number-wanting) (error))
        (cm:send-data consee free-addr *self-address* *address-size*
                      received-consee))
        ;; now the rendezvous point does its work.
        ;; anyone who received two messages (i.e. the received-conser
        ;;   and the received-consee bits are true) should do it
        (cm:set-context)
        (cm:load-context received-conser)
        (cm:load-context received-consee)
        ;; actually do the send
        (cm:send-data new-address conser consee *address-size* ignore-bit))
       ;; now restore the original context
       (cm:set-context)
       (cm:load-context temp-context))
```

Figure 2.6: A program to implement `cm:cons` in terms of `cm:enumerate`.

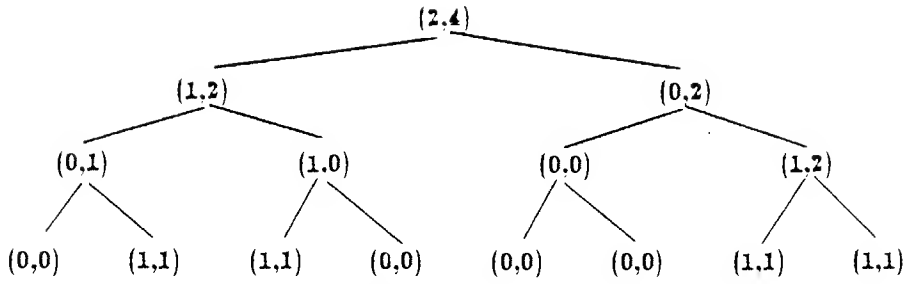

Figure 2.7: Enumeration by subcube induction.

typical `cm:send-data` instruction.

In general, the instructions which involve communication cost much more than the instructions which manipulate the local state of a processor, however there are a few exceptions, such as floating point operations, in which the local operation costs as much as a communications operation.

2.2.6 Virtual Processors

It is possible to 'time share' PE's to allow a larger set of 'virtual' PE's, each virtual PE having a smaller local memory and running correspondingly slower. There is microcode to support virtual PE's, and because we use less than 512 bits of the local memory to implement our simulator whereas there are 4096 bits of memory in each PE, we can run our simulators with half a million virtual PE's.

Chapter 3

Applicative Architectures

We simulate static data flow, VIM style dynamic data flow, and combinator reduction architectures. This section defines what we mean by each of these architectures and describes a high level implementation strategy for each.

3.1 Static Data Flow

A static data flow computer[Den74] consists of two active components:

- The program graph, and
- the structure storage.

3.1.1 Static Data Flow Program Graphs

Abstractly, a static data flow program graph is a directed graph with processing elements at the vertices. Data values travel along the arcs of the graph, and processing elements 'fire' when enough data is available on their input arcs, consuming data values from their input arcs and generating data values on their output arcs. We will simulate the program graph by statically allocating one connection machine PE to each data flow PE, and sending messages

among the PE's. The macro instruction sequence that drives the simulator will serially select each 'kind' of data flow cell (e.g. adder cells and then multiplier cells) and subselect those that have enough data to fire. Then the remaining set of PE's will be instructed to do some local arithmetic which is dependent on the 'kind' of the cell selected (e.g., add the two input numbers to create an output number for data flow 'ADD' nodes), and finally the output results will be sent as messages. The next 'kind' of data flow cell is then selected to run and we do the whole process again.

3.1.2 Static Data Flow Structure Storage

The structure storage of a static data flow system allows us to have data values which are too large to represent with one message packet, instead these large data values are stored on a heap, with pointers used to represent the values. We choose cons cells as our basic structure storage, and introduce a few new kinds of program graph cells.

Balanced trees of cons cells can be used to implement arrays with $\log n$ access latency (where n is the number of elements in the array). This is just a special case of the way that balanced m-way trees are used to represent arrays in the VAL simulator[AD79] or in VIM[DSG85].

Each cons cell will be dynamically allocated a PE on the CM, and the addresses of these cells will be passed around in the program graph. When the program calls for the CAR of a cons cell, the data flow graph obtained by compiling the program contains a CAR node. When the processor, M, which represents the CAR node of the data flow graph receives a message containing a pointer to a processor, N, representing a cons cell, M must extract the CAR of the cons cell represented by N. The details of how this extraction is performed and the impact on garbage collection will be discussed below. Our first design uses a high level software convention which copies a cons cell whenever too many pointers to the cons cell are needed. Our second design uses fan-in trees (which were introduced in Section 2.2.4). Our third and final design uses the `cm:get` and `cm:send-with-add` instead of `cm:send-data`, and

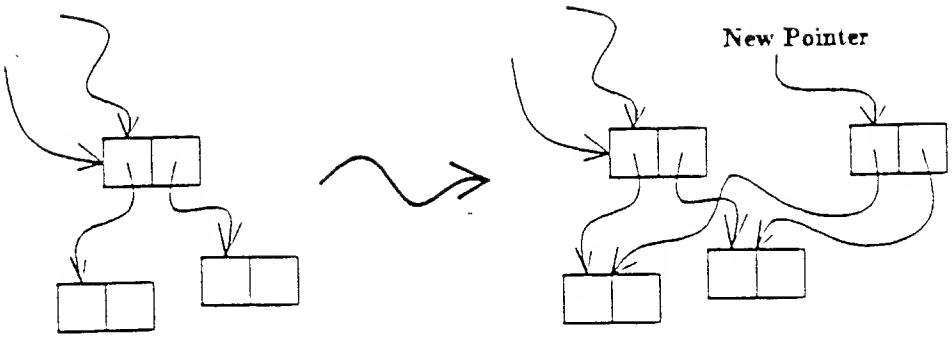

Figure 3.1: When a third pointer is needed to a cons cell, we copy the cons cell.

avoids most of the complexity of the first two designs.

Structure Storage with no Fan-In

Our first design for the static structure storage uses a high level convention which has the property that at most two pointers to a given cons cell exist at any time, and if more pointers are needed, the cons cells are copied, copying the data (See Figure 3.1). Note that since the data can be cons cells, we may end up consing up a lot of processors to represent relatively few cons cells (See Figure 3.2).

In this design, to implement CAR, a message will be sent to the processor representing the cons cell asking for the CAR, and the reply will contain the needed value. To implement garbage collection with this design, we introduce a kill message which says that a pointer is no longer needed. When both of the pointers to a cons cell have been killed, the cons cell can be deallocated. This is a very simple reference counting scheme, in which we can prove that the reference count for a cons cell becomes no larger than two. Thus, a pointer to a cons cell can be considered to be a capability to perform operations on the cell.

Since we do not support a REPLACE-CAR operation (we are, after all simulating architectures

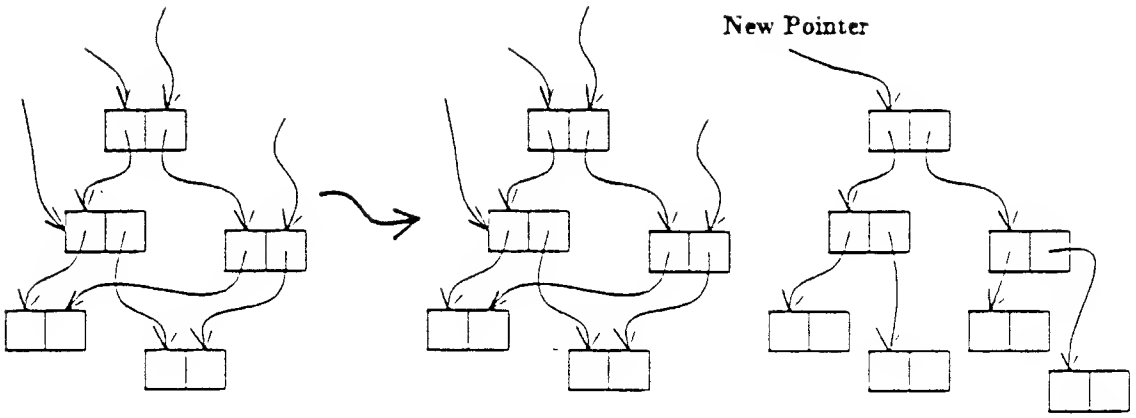

Figure 3.2: When copying a cons cell, the data needs to have the *copy* operation performed also. In this case, the whole tree needs to be copied because all the cons cells are 'saturated'.

which support *applicative* languages), we know that our structure storage graph is acyclic, and that the only important notion of equality between two subgraphs is the standard LISP EQUAL operation: Two objects are EQUAL if they are both atoms (e.g. integers), or if they are both cons cells and their respective CAR's are equal and their respective CDR's are equal. This definition of equality means that it is acceptable to copy a cons cell to handle the fan-in problem.

We want cons cells to have the following operations:

`cons` *car cdr*

Create and return a pointer to cons cell with *car* and *cdr* as specified.

`kill` *object*

If *object* is a pointer to a cons cell, then it will be an error in the future to use *object* to perform operations.

copy *object*

If *object* is an atom, then return *object*, otherwise, create a pointer to an object which is equal to *object*.

car *object*

If *object* is an atom then this is an error. If the CAR of *object* is an atom then return the CAR, otherwise return a pointer to an object which is equal to the CAR of *object*. An implicit KILL of *object* is performed by this operation. Thus, if the program needs to keep a copy of the original cons cell, and manipulate the CAR of the cons cell, first a copy should be done, and then a car should be done. The rationale for this implicit KILL is that a data flow CAR operator takes a token containing the address of a cons cell on the heap, and creates a token containing the CAR of that cons cell, destroying the original token in the process. Thus, it makes sense for there to be an implicit KILL to be performed when the CAR is taken. If a program needs to keep a copy of a pointer to a cons cell, then the corresponding data flow program will make sure to have made a copy of the pointer to the cons cell before passing the pointer to the CAR data flow node.

cdr *object*

Similar to the CAR operation, except it operates on the CDR of the object.

appendcar *object newcar*

If *object* is not a pointer to a cons cell then this is an error. Otherwise, return a pointer to a cons cell with CAR equal to *newcar* and CDR equal to the CDR of *object*.This operation implicitly kills *object* and *newcar* in order to allow efficient implementation. In other words, the "caller" gives up the capability for *object* and *newcar*, and gets back a capability for a new cons cell. The returned cons cell is, in general, a new cons cell, but if the implicit kill operation would deallocate the old cons cell, we can reuse it. This reuse is called *appending in place*.

There is a corresponding APPENDCDR operation which returns a pointer to a cons cell with the CDR modified.

Our main motivation for introducing this capability based system is that the message delivery mechanism provided on the CM does not allow multiple messages to be simultaneously delivered to a single PE. By introducing the capability system we can guarantee that some statically bounded (say two) number of pointers to any given PE exist at any given time. Thus, if our static bound is N (in this case $N = 2$), we can add $\lceil \log N \rceil$ bits of information, called the *in-box number* to our capability pointer and avoid message collisions. We will guarantee that for any in-box number i and any PE number p there will be at most one capability pointing to in-box number i on PE number p, and we can treat a capability as a permission to write into a particular in-box. During any message send, we will send only messages which have the same *in-box number*, and thus there will be no message collisions.

Thus, when an cons cell is created, we create a capability with its *in-box number* equal to zero, and every time we need to *copy* a capability, we might perform the following algorithm:

- If some in-box number is unallocated, then we pick such an in-box number and return a capability with that in-box number.
- If every in-box number is allocated, then we recursively COPY the CAR and the CDR of the object, and CONS up a new object with that CAR and CDR.

When we KILL a capability which refers to a cons cell, we merely note that the in-box number of the capability is free. If all the in-box numbers are free, then we can KILL the CAR and CDR of the object, and return the PE in which the object resides to the set of free PE's.

When we take the CAR of an object, we can note whether the reference count of the object is one. If it is, we can simply return the CAR of the object, KILL the CDR of the object, and return the PE in which the object resides to the set of free PE's. If the reference count of the object strictly greater than one, then we decrement the reference count and return a COPY of the CAR of the object. Taking the CDR of an object is similar.

When we APPENDCAR an object, if the reference count is one we can KILL the old CAR of the object, and store the new car in the objects CAR location, returning the original capability. If the reference count is greater than one, then we copy the CDR of the object, and CONS up a new cell using the new CAR and the copy of the CDR. Performing an APPENDCDR is similar to performing an APPENDCAR.

Thus, by designing our system to use a capability scheme (and enforcing the scheme by software conventions), we gain the following advantages:

- We avoid message collisions at the destination PE's.
- We have garbage collection (using reference counts).
- We can optimize certain operations (such as APPENDCAR when the reference count is one). For applicative programming languages, this can be an important optimization[DSG85].

There are two often cited disadvantages of reference counted garbage collection which[Den80] rebuts for the case of applicative archictures in general. We believe our system also deals with these alleged disadvantages satisfactorily:

- Reference counting does not work for cycles. However, because we are implementing an applicative system, our structure graphs can never have cycles, and this objection is unimportant.
- Reference counting is expensive. However, given the nature of the CM message delivery scheme, we need some mechanism to prevent multiple messages from arriving at the same place at the same time. Our scheme is to keep the reference counts of PE's very low, and once we have implemented this scheme, most of the expense of reference counting (i.e. the incrementing and decrementing of the reference count, and dealing with overflow of the reference counter) are dealt with. On a serial machine, there are some real-time considerations: In general, we can not predict how many cons cells will be freed by a single KILL operation, and thus the time to perform a KILL can take

arbitrarily long on a serial machine (unless one implements a mechanism whereby some of the work involved in the KILL operation is defered). On a parallel machine, the PE which sent the KILL message does not need to wait for the KILL operation to complete before continuing, and so no additional waiting is incurred.

There is one problem with the algorithm that we have specified, and that is that it is not very efficient in space. On a serial computer with no reference counts, it always takes $\Theta(n)$ memory locations to represent a structure storage containing n cons cells. With the algorithms we have specified, the number of PE's required to represent n cons cells is also dependent on the number of copies of the graph we have made, and the order in which we have made the copies.

For the following theorems, we will assume without loss of generality that the maximum reference count for a PE is two.

Definition 2 *A PE is* saturated *if there are two pointers pointing at the PE. A graph of PE's is saturated if all the PE's in the graph are saturated.*

Theorem 1 *Using the algorithms specified above, in the worst case, it takes $\Omega(nm)$ PE's to represent n cons-cells with m pointers pointing 'in' to the cons cells from the 'outside' of the structure storage (e.g. from the program graph).*

Proof: First we construct a balanced binary tree containing $n = 2^k - 1$ cons cells (where k is the depth of the tree). For the worst case analysis, assume that all of the PE's representing the cons cells in the tree are saturated, and that there are two pointers from the 'outside' (we can easily force this to be the case by constructing the graph in the right order). Let X be one of the two pointers to the head of the tree from the outside. If we perform a COPY operation on X, we will need to copy the entire tree, because when we need to copy the PE at the head of the tree, it will already be saturated, and we will need to cons up a new cell, copying the CAR and the CDR of the cell. The same thing will happen while copying the CAR and the CDR of the cell because they will be saturated. Thus the entire tree will be copied.

(See Figure 3.2 for an example of how this behaves.) If we COPY X again, the entire tree will be copied again, and thus if we make m copies of X, we will need mn PE's to represent the graph. Thus in the worst case, we need at least mn PE's to represent m pointers to n cons cells. □

In fact, we can construct a worse lower bound.

Theorem 2 *Using the algorithms specified above, in the worst case, it takes $\Omega(m2^n)$ PE's to represent n cons-cells with $O(m)$ pointers pointing to the cons cells from 'outside' of the graph.*

Proof: Suppose that the atoms at the leaves of the tree in Theorem 1 are all the same (See Figure 3.3). In that case there are only k distinct cons cells in the tree (by our definition of equality). The tree can be constructed with only k explicit *CONS* operations, again by construction, and it will take $O(2^k)$ cells to represent the tree with only one copy of it (since the leaf of the tree (there is only one distinct leaf) will have 2^{k-1} pointers pointing at it, it will need at least 2^{k-2} cells to represent it). Then when we make m more copies of X we will have consumed $\Omega(m2^n)$ PE's to represent m copies of a structure with only n distinct cons cells. □

And finally, we can show that the algorithms specified above are not always bad, i.e. there are cases where the algorithms run well.

Theorem 3 *Using the algorithms specified above, in the best case it takes $\Theta(n+m)$ PE's to represent a graph with n distinct cons-cells and m pointers from the 'outside' into the graph. This best case can actually be realized.*

Suppose that we have a balanced binary tree with $n = 2^k - 1$ distinct cons cells represented by $2^k - 1$ PE's, each with reference count equal to one. We can make m copies of the tree by the following algorithm:

- Let X_0 be the head of the tree.

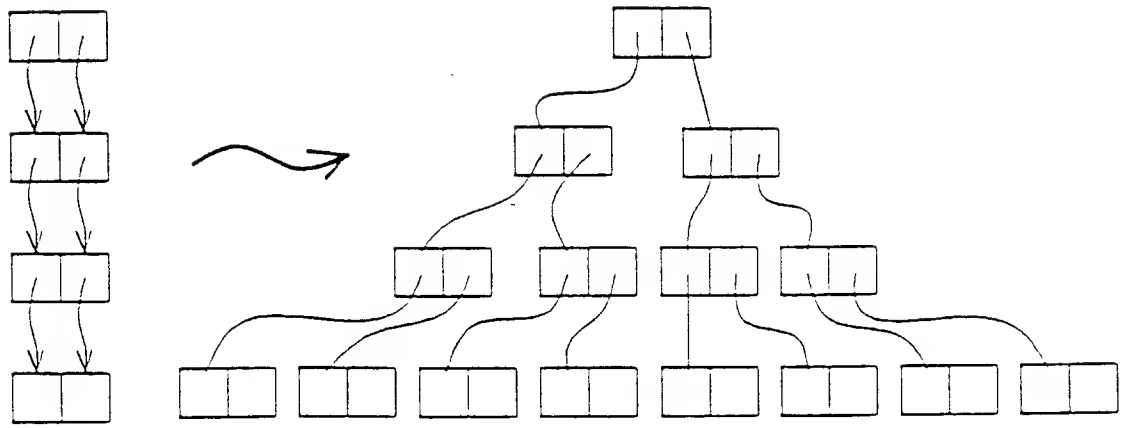

Figure 3.3: If all the leaves of the tree are equal, it can take $\Omega(m^2 n)$ PE's to represent m copies of a structure with only n distinct cons cells.

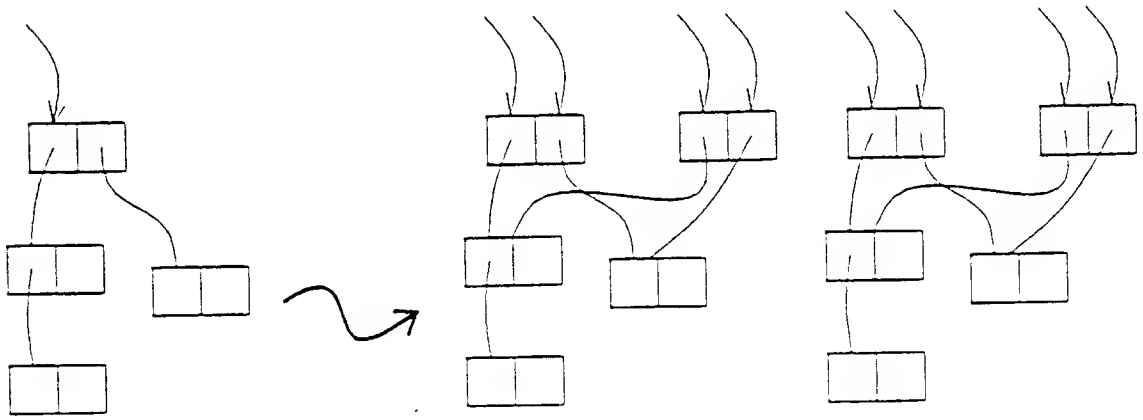

Figure 3.4: In the best case it takes $\Theta(n+m)$ PE's to represent a graph with n distinct cons cells and m pointers into the graph.

- For each i in $\{1, \ldots, m\}$ let X_i be the copy of X_{i-1} (See Figure 3.4).

Now we will need a few definitions and a lemma to show that the above algorithm works.

Definition 3 *If we have a single pointer, p, into a graph of cons cells, then a PE, r, which is part of the representation of that graph is* on level i *if there is a way to take* $i-1$ CAR*'s and* CDR*'s from the pointer, and end up at r. (For example the top cons cell is on level one.) (Note that we mean the abstract operations of* CAR *and* CDR *rather than the implementation of* CAR *and* CDR *which can modify the representation, in the structure storage, of the objects.*

For complete balanced binary trees the *level* is clearly unique.

Definition 4 *Given a pointer, p, into a complete balanced binary tree of cons cells, p is called* k-loaded *if all PE's on level* $l \leq k$ *have reference count equal to two, and all PE's on level* $l > k$ *have reference count equal to one.*

Note that our definition of *k-loaded* is very restrictive. There are binary trees which are not k-loaded for any k. It turns out that the binary trees which we will build in the following proofs are k loaded.

Lemma 1 *Starting with a complete balanced binary tree of distinct cons cells of depth k represented by* $2^k - 1$ *PE's each with reference count equal to one, the above algorithm will consume* 2^k *additional PE's during the* $m = 2^{k+1} - 1$ COPY *operations, and* X_m *will be k-loaded, and the PE's on level k and higher will be shared with* X_0*, and if the complete balanced binary tree is not the whole graph reachable from* X_0*, any PE's on level* $k+1$ *and higher will have the same reference count they started out with.*

Proof of lemma: Inductively on k.

When $k = 0$, $m = 1$, and we can make one copy of the tree without consuming any additional cons cells (since the reference count of the head was one.) After copying, X_1 will have the property that only the top cons cell (at level one) will have reference count equal

to two, and the rest of the PE's will have reference count equal to one. All the PE's are still shared with X_0, and any PE's on level one or higher have not been touched so their reference counts are the same.

If the lemma is true for $k = l$, we need to show that it is true for $k = l+1$. During the first $a = 2^l - 1$ copies we get that X_a is l loaded by the inductive hypothesis. Thus creating X_{a+1} will force all the PE's in the first l levels to cons up new PE's which will all have reference count equal to one. The PE's on level $l+1$ will have their reference counts equal to two, since the cons cells are all distinct and thus every PE on level l will have increased the reference count of a distinct PE on level $l+1$ (and the reference counts on level $l+1$ were one until this point). If we then perform the next a copies to create X_{2a+1} we know by the inductive hypothesis that the PE's on level $l+1$ still have their reference counts equal to two, and X_{2a+1} is l-loaded, so X_{2a+1} is actually $l+1$-loaded. The PE's on level $l+1$ or higher are still shared, and any PE's on level $l+2$ or higher have their reference counts unmodified. □

Lemma 1 thus proves Theorem 3. □

It is important to realize that no matter what we do, if we have a complete balanced tree of depth n, with only n distinct cons cells in the graph, we will be forced to use $\Omega(2^n)$ PE's to represent the graph because of the fan-in limitations that our methodology requires. Thus, for this case we can not expect to do very well, and it is not fair to judge our algorithm on the $\Omega(m2^n)$ bound derived in Theorem 2, but instead to judge our algorithm on the $\Omega(mn)$ bound derived in Theorem 1. The reason that this is important, is that we will now describe an algorithm which allows us to achieve much better worst case bounds for the tree of depth n with 2^n distinct cons cells, but the improvement will still by swamped by the exponential term for the case of the tree of depth n with only n distinct cons cells.

Fan-In Trees

We can improve on the worst case bounds given above, making them essentially the same as the best case bounds of the above algorithm by introducing *fan-in trees* into our design.

First we modify our specifications a little bit in order to make our implementation a little easier. We require that the COPY operation return two pointers and that it performs an implicit KILL of its argument.

The algorithm: Structure storage is represented by two classes of PE's. The *cells* and the *fan-ins*.

A *cell* behaves in approximately the same way that a cons cell PE behaved for the last algorithm, except on a COPY p operation: If the reference count is two, then the cell conses up a *fan-in* PE which will 'forward' references to p. The two returned pointers are pointers to the fan-in PE's in-boxes.

A *fan-in* PE acts like a buffer, 'protecting' its forward pointer from requests. The fan-in PE has all of the reference counting machinery of a cons cell, plus a location to cache the CAR and CDR of its protected cons cell. As soon as the fan-in cell has cached both the CAR and the CDR of its protected cell, the fan-in PE changes its 'type' into a *cell* PE.

In order to make this caching work, we introduce two new operations called CAR-KEEP and CDR-KEEP which are just like CAR and CDR respectively, except that they do not perform an implicit KILL on their arguments.

When a fan-in PE, p, receives a

KILL, The *fan-in* PE decrements the reference count, and notes which of its 'in-boxes' is freed by the KILL. If the reference count drops to 0, then the PE sends a KILL message to its 'protected' pointer and returns itself to the free pool.

COPY p, The *fan-in* PE checks to see if it has a free in-box. If so then it returns p and a pointer to the free in-box, noting that the in-box is no longer free. If there is no free in-box then it creates a new *fan-in* PE which 'forwards' to p, returning two pointers, both of which point to the new *fan-in* PE.

CAR p The *fan-in* PE checks to see if it has cached the CAR. If not it issues a CAR-KEEP request to its protected cell and caches the reply. At this point, the CAR has been cached. If the implicit kill of the p (the fan-in cell) decreases the reference count to zero, then we

can simply return the CAR and kill the cached CDR if it is present. If the reference count is still positive, then we send the CAR a COPY message, caching one of the results in its own CAR and returning the other result.

CDR p, The *fan-in* PE behaves similarly to the way it did for CAR p.

CAR-KEEP p, The *fan-in* PE behaves the same as it did for CAR p, except that we do not perform the implicit KILL, so we always have to perform a COPY of the cached CAR.

CDR-KEEP p, The *fan-in* PE behaves similarly to the way it did for CAR-KEEP p.

APPENDCAR p c, If the *fan-in* PE has reference count equal to one, then it sends a KILL message to the cached CAR if it is present, and caches c as the CAR. If the fan-in cell has reference count greater than one, then it conses up a new cons cell, storing c in the CAR of the new cell, and its CDR (either obtained by performing a COPY of the cached CDR if it is present, or else by sending a CAR-KEEP to its protected cell) in the CDR of the new cell. The new cell is returned, and the reference count of p is decremented (which never results in a KILL message being sent to its cached CDR because in this case the reference count was not one.)

APPENDCDR p c The fan-in PE behaves similarly to the way it did for APPENDCAR p c.

Theorem 4 *In the worst case, the above algorithms consume at most $O(1)$ PE's and $O(\log n)$ time for each operation (i.e. CAR, CDR, COPY, etc.), where n is the size of the tree being operated on.*

Proof: We will do a case analysis on the operation:

COPY, At most, one new fan-in PE is consed up, which takes constant time.

KILL, Consumes no PE's. An arbitrary number of PE's may be freed by any particular KILL operation, but there is no need for the sender of a KILL message to wait until the KILL is completed before continuing with useful computation. Even so, the arbitrary amount of

computation consumed by a KILL operation is inherent in any reference counted scheme, and we depend on a parallel model of computation in order to justify not counting this expense as part of the KILL. If we were to amortize the serial cost of the KILL over all the KILL's, COPY's, and CONS's we could easily show that KILL's are cheap 'on the average'. (The cost of performing all the KILL's is $O(c)$ where c is the number of CONS's and COPY's, and so the average cost of any particular operation is $O(1)$.)

CAR-KEEP and CDR-KEEP, A COPY is performed, which consumes at most one PE and takes constant time. Actually taking the CAR takes at most $O(\log n)$ time.

CAR and CDR, May perform the corresponding KEEP operation (which consumes at most one PE and takes constant time), and then either a KILL (which takes constant time and consumes no PE's), or a COPY (which also takes constant time and consumes at most one PE). (Thus we may consume up to two PE's during a CAR or CDR.)

APPENDCAR and APPENDCDR, Either operation performs

- a KILL, or
- a cons, followed by either a CAR-KEEP or a COPY.

The most expensive path is to perform a cons (consuming one PE) followed by a CAR-KEEP (consuming one more PE). □

The fan-in tree has the property that it smooths out the worst case behavior for copying pointers at the expense of making the CAR operation slower: The CAR operation was very fast when cons cells were copied since the PE pointed to always contained the data. The CAR operation could take much longer in the system described in this section because the request for the CAR must propagate through the fan-in tree. Note, however, that the CAR is cached so that the expected time for performing a CAR operation improves as more CAR operations are performed.

Structure Storage using cm:get and cm:send-with-add

With the introduction of the cm:get and cm:send-with-add CM macro instructions, our design for the structure storage becomes much simpler. The CAR operation is implemented by doing a cm:get operation. In other words, we can freely propagate copies of pointers through, without worrying about what happens when several processors try to fetch the CAR of a cons cell at the same time.

Garbage collection becomes a little trickier however. A reference counted garbage collection scheme would work, just as it did for the fan-in tree implementation, however, since this design does not require that reference counts be maintained just to keep track of how to copy pointers, we might decide on another garbage collection technique, such as *mark and sweep* (which was briefly explored in[Chr84]). We decided to use reference counting garbage collection, partly because we understand it, and partly because it helps to maintain similarities between the various designs for the structure storage.

Given these decisions, the structure storage has a very simple design. We use cm:get to perform the CAR and CDR operations. We use cm:send-with-add to increment and decrement the reference count. The APPENDCAR operation can be implemented by doing a cm:get on the CDR, and incrementing the CDR's reference count, then doing a cm:cons to get a new PE, and storing the new CAR and the 'copied' CDR in the new PE.

The fan-in trees are still implicitly being used, but because the fan-in trees are implemented at a very low level, the implementation of our simulator becomes much more simple, and runs much more quickly.

3.2 VIM Style Dynamic Data Flow

As Will Rogers would have said, 'there is no such thing as a free variable.'

Another proposed data flow architecture, which we will refer to as VIM style dynamic data flow, is described in[DSG85]. VIM allows functions to be treated as first class data values:

They can be passed to functions and stored in structures. One result of this treatment of function values is that recursive programs can be written[Kus84]. The VIM system was originally designed to be run on a MIMD system such as a network of lisp machine processors. This section provides an abstract description of VIM, and describes the requirements for the 'primitive objects' that we will use to simulation VIM. We conclude this section with a description of our simulation.

3.2.1 What is VIM

The VIM system allows functions to be treated as data, and provides several other facilities such as early completion structures and guardians[DSG85]. Given the mechanism that we will develop in this section, it would be fairly straightforward to implement both early completion structures and guardians on our data flow simulators, but we have not done so.

We will thus consider only the treatment of functions as data. In applicative languages data values are immutable, and so in VIM style dynamic data flow functions as data are immutable. On the other hand data flow graphs are mutated while being interpreted (i.e. the state of the program is in the data flow graph). This means that we need to distinguish between functions as data and functions as data flow graphs. As proposed in[DSG85], VIM uses function *templates* to represent the function value as a datum. To apply a function to some arguments requires that the template be copied into otherwise unshared memory and 'transformed' into a data flow graph. The resulting data flow graph will have been dynamically allocated from the memory heap, and will eventually need to be deallocated.

The VIM system is defined as an abstract machine rather than a programming language. A compiler is needed to translate an applicative program into a data flow graph. Thus, to support recursive programs in an applicative language it is sufficient to provide functions as data values, since recursive programs can be implemented by passing functions as values[Kus84].

All functions in our world will be unary, i.e. they take exactly one argument. To simulate functions of more than one argument, we can either curry functions to achieve multiple ar-

gument functions, or we can package up the arguments into an structure (from the structure storage) and pass them all as one argument.

In general, we expect that several apply nodes will be trying to apply the same function to different arguments at the same time. In particular, there may be n apply nodes trying to make a copy of a graph which consists of m PE's at the same time. We would like this graph copy to run quickly, and that is the main difference between implementing static data flow and VIM style dynamic data flow.

Note that in VIM, any particular instance of a data flow node will fire at most one time (because all loops are implemented by using recursion). In particular any instance of an apply node will only fire once. Thus, when the apply node 'splices' the destination address for the function's result into the function's graph, there is no problem with values arriving at the destination out of order.

3.2.2 The Primitives Used to Simulate VIM

Our simulation of VIM allows functions to be treated as data values by dynamically allocating storage for the function, and providing an apply operator to create a new copy of some subgraph of the program graph. The apply operator can be part of a data flow graph just like any other operator, such as the add operator.

In order to support lexically closed function values, we introduce another data type, the *closure*. By the time the compiler has translated an applicative program into a data flow graph the variables *per se* are gone. A closure is a value which can be combined with another value (the value being "bound" to the "variable") to yield some other value (either a closure value, or a function value, in which the "variable" is "bound"). The compiler is then able to translate the functional programs into data flow graphs. See Figure 3.5 for an example of this translation.

To support closures, we introduced a new data type called *closure*. We define the CLOSE operation, and reference count maintenance operations on *closure* objects (and as usual these

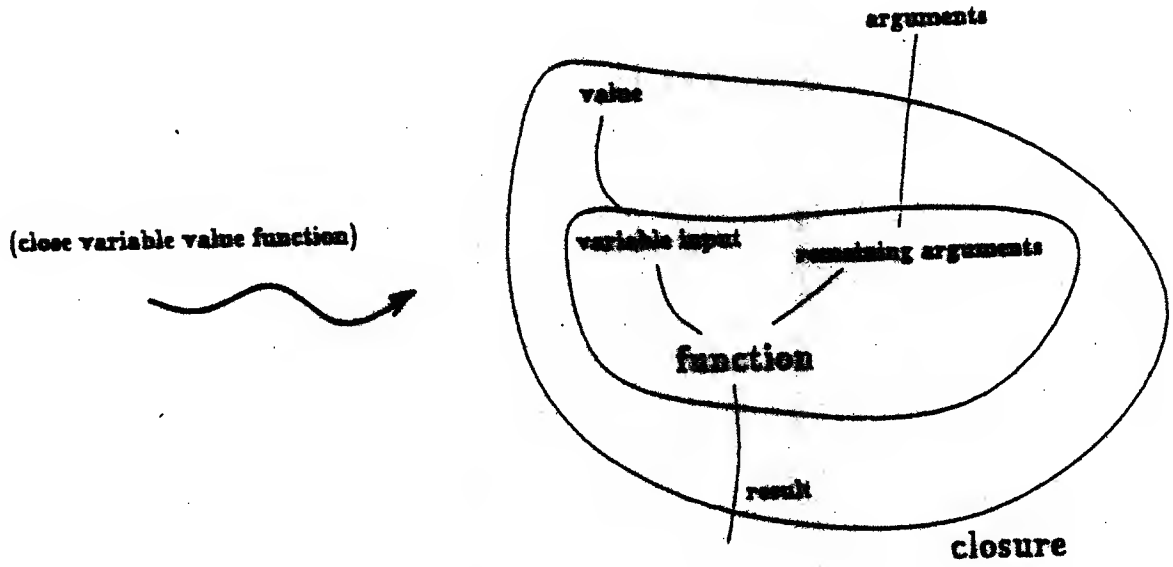

Figure 3.5: Translating lexically scoped functions into closures.

operation manifest themselves as data flow operators). The CLOSE operation may involve performing some copying. The reference count manipulation is done as for cons cells.

We need a scheme to deallocate closures and function values when they are no longer needed, possibly either by maintaining a reference count on the data flow graph itself, or by causing the graph to deallocate itself automatically when it has completed its computation, as in[DSG85].

3.2.3 Implementing the VIM Primitives

In order to simulate VIM style dynamic data flow we need to design a representation for data flow graphs which can be quickly transformed from a template into a data flow graph which can run on the machine. Our implementation uses just one representation both for the function as a datum (i.e. the template) and the function as a data flow graph being interpreted on the machine. The transformation from template to data flow graph is thus trivial, the only requirement being that the graph be copied so that the new copy is not shared with any other part of the system (it will then be safe to mutate the graph).

The design we settled on is the following: A function value is represented by a set of PE's. The function graph deallocates itself using a *release* mechanism, as described in[DSG85]. Each PE represents an operator in the data flow graph of the function. The PE's have two graphs superimposed on them. The first graph is the directed graph which corresponds to the data flow graph of the function. The second graph is a balanced binary tree which contains all of the PE's representing the function.

The data flow graph is used just like the data flow graph was used for the static data flow implementation. Our function objects understand only one operation, the APPLY operation. It turns out that we do not need a KILL operation, since the function templates can be kept around in the machine effectively forever (we are simulating a system which runs for a short period of time, and so we do not need to worry about deallocating the template), and copies of the template are made only during the APPLY operation, in which case the function will

deallocate itself with the *release* mechanism.

The tree which is superimposed on the PE's is used to perform operations on the graph. The tree is a balanced binary tree, where the head of the tree knows how many vertices, x, are in the tree. The tree is used to perform operations such as the copy during the APPLY operation because a request can be propagated to the PE's in time $O(\log x)$.

As stated above, we expect that many instances of each function will be applied in parallel. Suppose that n apply cells receive their data and can fire at the same time, and that they all receive the same function, f, to apply. Suppose further that there are m PE's in the template of f. Given fixed m, we require that any design be able to perform the application in time faster than linear in n, because otherwise the parallelism that is present in the program will be lost. We also would like to be able to apply different functions at the same time, and that the speed of copying be a slowly growing function of m (the number of nodes in the graph), even though both the number of different functions and the maximal m are fixed once the program has started running. The following design can apply n copies of a graph with m nodes in time which is polylogarithmic in n and m.

A useful subroutine is the operation of making one copy of a template. This can be done by first identifying the graph (propagate the fact that a copy is to be made through the tree which is superimposed on the template PE's), and then by using the algorithm to copy a graph described in Section 2.2.5.

The algorithm is as follows (See Figure 3.6):

- First the n apply cells must arrange themselves into a binary tree. This can be done using `cm:build-tree` (which is described in Section 2.2.4).
- The root of the tree makes one copy of the template for itself, and designates itself a *copy-site*. All the PE's which are not at the roots of their trees are not *copy-sites* to start with.
- While there is a *copy-site* do the following: (Note that the `cm:global-or` operation can be used to find out if any PE's are currently *copy-sites*.)

- For each *copy-site*, p, do in parallel:
  - If p has a left child, then make a copy of the graph, and send a pointer to the head of the copied graph to that left child. That left child becomes a *copy-site*.
  - If p has a right child, then make a copy of the graph, and send a pointer to the head of the copied graph to that right child. That right child becomes a *copy-site*.
  - Remove p from the process by making it no longer a *copy-site*.

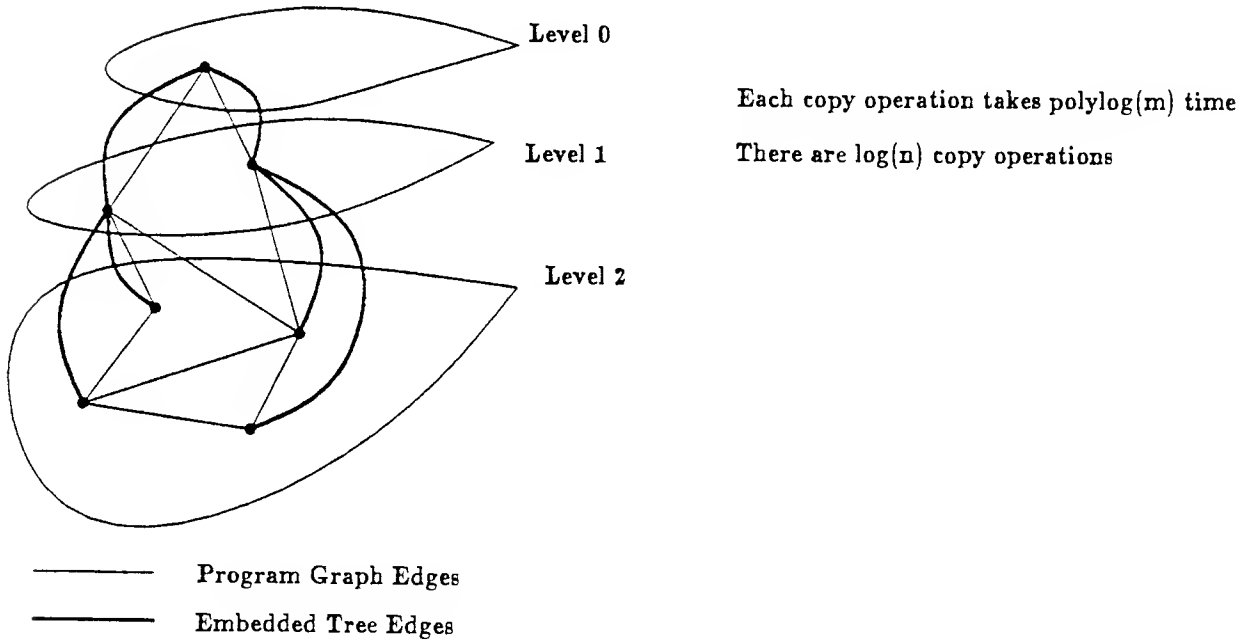

Figure 3.6: It is possible to make n copies of a graph with m vertices in time which is polylog in n and m.

Since the tree is roughly balanced, the while loop will run a polylog number of times in n, and the operation of making a single copy of graph with m vertices takes $\log m$ time.

The APPLY operation takes a function object and a value, copies the function object to create a copy which is not shared by any other part of the system, and modifies the graph to start it running: The argument is placed into the 'begining' graph, and the destination of the APPLY node is placed at the 'end' of the graph. The graph is then 'turned on' and starts running just as it did in the static data flow simulation.

3.3 Combinator Reduction

We simulate Traub's Abstract Combinator Reduction Architecture[Tra84] on the Connection machine by allocating one PE to each combinator *application* cell, and then sending the appropriate 'reduce', 'reduce-ack' and other messages between the PE's. Traub's architecture is particularly easy to simulate on the CM because it is easy to map combinator cells on to PE's with very few design decisions.

We will describe the implementation of a combinator reduction system that performs the following reductions.

$$
\begin{aligned}
(I\ x) &\Longrightarrow x \\
((K\ x)\ y) &\Longrightarrow x \\
(((S\ f)\ g)\ x) &\Longrightarrow ((f\ x)\ (g\ x)) \\
(+\ x\ y) &\Longrightarrow x + y
\end{aligned}
$$

We have chosen the above reductions because they include one mathematical operator (which is strict in its arguments) and the basic combinators from[Tur79]. Once the implementation strategy for these combinators has been explained it should be easy to understand the implementation strategy for richer sets of combinators.

Conceptually there is only one kind of cell in Traub's architecture, the *application* cell. The *application* cell behaves as follows:

There are two messages that the *application* cell recognizes: The `reduce` message, and the `increment-reference-count` message. The *application* cell has two variables in addition to the machinery for reference counts, the `function` and the `argument`. The *application* cell also

caches the result of the reductions that it performs.

When an *application* cell receives a `increment-reference-count` message, which contains a signed integer i, the reference count is incremented by i. Note that if i is negative, the reference count is decreased. If the reference count is decreased to one, then the *application* cell can be deallocated, and the reference counts of its `function`, `argument`, and any cached values can all be decremented.

When an *application* cell receives a `reduce` message it behaves as follows:

- If the cell has already performed a `reduce` then it will have cached the result. The cached result is returned to the cell requesting the reduction.

- Otherwise, we need to reduce the `function`. If the `function` is an atom (i.e., an I, K, S, $+$, or a number) then it is already reduced, otherwise we send it a `reduce` message and wait for the reply. If the reduced `function` is

  I We reduce the `argument`, cache the result and return it.

  K **or** S **or** $+$ We return the pair consisting of the reduced `function` and the unreduced `argument`. We cache that pair for future use.

  $(K\ x)$ We can throw the `argument` away, so we send a `(increment-reference-count -1)` message to the `argument`, and cache x as our result, and return x (note that we have to increment the reference count of x.)

  $(S\ x)$ Return the triple consisting of S, x, and the unreduced `argument`.

  $(+\ x)$ Reduce x and the `argument` in parallel and then add the results together, caching and returning the sum.

  $((S\ f)\ g)$ Cons up two new cells. Initialize the first new cell to contain f as its `function` and one of the copies of our `argument` as its `argument`. The first new cell will be our new `function`. The second new cell should be initialized so that g is its `function` and the second copy of our `argument` is its `argument`. The second new cell will be

our new **argument**. Now we go back and act as if we had just received a **reduce** message.

anything else is an error.

We need to be careful to correctly update the reference counts, and update them in the correct order. For example, it would not be correct to decrement the reference count of an object and then increment the reference count of another object, because if the object being decremented happened to be the same as the object being incremented, we could end up deallocating something we did not want to deallocate. If our decrements and increments appear in the correct order in our program, we are guaranteed that there will be no race conditions which cause improper deallocation of resources (since the CM is a SIMD machine).

Traub[Tra84] proves that the above scheme will work without errors, assuming that the original combinator program has no type errors. Also note that the only parallelism that this architecture allows is in the reduction of arguments to strict operators. It may be possible to achieve better performance by allowing the reduction of other expressions to proceed in parallel.

Chapter 4

StarTalk

In this chapter we provide motivation for a special purpose language to simulate MIMD systems on the CM. We describe our language, which is called StarTalk, and discuss the optimizations that our compiler performs.

Simulating MIMD machines on the CM is difficult without a special purpose language because the programmer is forced to write unstructured code with explicit labels. E.g., our original specification for cons cells was relatively small (about a page long), but after manually translating it into CM macro instructions with explicit manipulation of 'state' values (i.e. the 'program counter' of the MIMD processor which we are simulating), the code had grown by more than a factor of ten. Furthermore, the original structure of the code had been lost, and because this process was expensive, when changes needed to made in the code, the temptation was great to modify the 'object code' instead of modifying the source code and manually recompiling. There are also a large class of optimizations which are infeasible to perform by hand. By designing a special purpose language we were able to maintain a compact and readable specification of our simulator, and perform optimizations.

The StarTalk language is 'object oriented' in that the program specifies what any particular object does. There are primitives to deal with message passing and local arithmetic, as

well as flow of control structures such as LOOP and IF. There is a static subroutine mechanism which allows non-recursive subroutines to be written. The STARTALK language is very much like CL1[Baw84] in that one thinks about the objects and how they behave. In CL1, it was absolutely necessary to avoid propagating copies of pointers to objects because the machine programming model used by CL1 includes only a very simple message passing primitive (i.e. it includes `cm:send-data` but not `cm:send-with-logior` or `cm:get`). In STARTALK, one need not necessarily worry about the propagation of copies of pointers to objects.

Here we will describe the implementation of STARTALK, including the flow of control, the subroutine linkage mechanism, and the 'interpreter' cycle.

The STARTALK compiler translates STARTALK programs into CM macro code by the following conventions. The compiler identifies basic blocks of code (i.e. code which is executed linearly) and gives each basic block a unique positive integer identifier in a compact interval starting with zero. Every processor has a field which can be thought of as the 'program counter' for that processor. When that field contains number i in a given processor, that processor 'wants' to run basic block number i. It is the responsibility of the code in basic block to set the 'program counter' for that processor to the block number of the next basic block which is to be executed. This value can be set to one of several values to simulate conditional statements, or it can be used to simulate loop statements. In fact, since the block number is simply data, the block number can be stored or computed in an arbitrary manner and used to simulate 'indirect' jumps. The subroutine linkage mechanism uses this to an advantage. Since STARTALK subroutines are static (i.e. there is no recursion or mutual recursion) it is possible to statically assign a location to be used for subroutine linkage. This location can be used to pass arguments, return results and store the program counter for the next basic block to be executed after the subroutine returns.

To interpret a compiled STARTALK program, the host computer can step through all of the basic block numbers, selecting the processors which 'want' to run a given basic block, and then broadcasting the instructions for that basic block. The broadcasted instructions will

have set the program counters for those processors to new values and when the interpreter cycles to the other block numbers the processors will make more progress.

It is possible to perform optimizations to improve the quality of the code. None of the optimizations descibed below are present in the current version of the STARTALK compiler.

Variable allocation can be optimized by doing global data flow analysis on the code. Local processor memory is a scarce resource on the connection machine. By performing global data flow analysis, it is possible to reduce the amount of memory actually needed to store the state of a STARTALK object. By reducing this memory requirement, we can run with more virtual processors and support a larger number of active objects at one time. This optimization was actually implemented, but is not used in the present version of the STARTALK compiler because we were more interested in quick prototyping than running large systems.

Common subexpressions can be factored out, especially the most expensive common subexpressions, into subroutines. For example, if one set of PE's, A, needs to perform a floating point multiply, and another set of PE's, B, needs to perform a floating point add, the STARTALK compiler can arrange for the normalization phase for both A and B to be carried out in parallel, by doing the actual multiplication while A is selected, and the actual addition while B is selected, and then normalizing with both A and B are selected. Currently, it is necessary for the programmer to explicitly place code into subroutines if such sharing is to be achieved. The cases where we actually use subroutines to improve the efficiency are code segments containing message sending code and code such as as `cm:cons`.

The object code generated by STARTALK looks bad. We suspect that a large improvement could be made by performing local optimizations, such as peephole optimization, and handling special cases such as when values are constants more consistently.

Chapter 5

Conclusion

In this chapter we will describe and analyze the preliminary measurements of the performance of our simulators, and conclude with a discussion of possibilities for further work and exploration of remaining open questions.

We have implemented two of the simulators described in this paper, The static data flow simulator, and the combinator reduction simulator. The VIM style dynamic data flow simulator has not been implemented. Our experiences confirm the widely held belief that programming parallel machines is not easy: Both the implementation of the STARTALK compiler, and the simulators in STARTALK were nontrivial, and our initial attempt to implement the simulators directly in the connection machine primitives was so difficult that we could not get it to work at all. One of the reasons that we have not implemented the VIM style dynamic data flow simulator is that these simulators are relatively difficult to write.

5.1 Performance of the Simulators

We would like to say that our simulators give *high performance*, but it is difficult to define what we mean by 'high performance' on simulators, especially when comparing hardware

designed especially for an architecture to the simulation of the architecture. This is especially true because it is always possible that our simulators are not as good as they could be, (e.g. it is possible that we have not done as good a job as we might have) which means that we can only give a good lower bound on the performance of the connection machine as a simulator.

We could measure several aspects of our simulator's efficiency, including the percentage of PE's which are being used at any time, the number of primitive operations per second (i.e., for static data flow we count the number of node firings per second, for dynamic data flow we also count the number of function applications per second, and for combinators we count the number of reductions per second), or the speed to run certain programs.

We chose to measure the number of primitive operations per second for several essentially serial programs. The reason that we chose to measure serial programs is to explore the *worst case* behavior of our simulators. Clearly if a program has much parallelism, we will achieve more primitive operations per second than if the program has little parallelism. The goal of this paper is *not* to explore the improvement to be gained from exploiting the parallelism in programs, but is rather to try to understand how to simulate parallel applicative architectures on the connection machine. We therefore decided to measure essentially serial programs in order to be able to compare our simulators to other implementations of applicative architectures.

There are several reasons that the decision to measure serial programs might be a bad one. The most important such reason is that if we were to compare the simulation of essentially serial programs on the connection machine to the simulation of the same program on a fast serial machine, then our results would be misleading. The serial machine will clearly have an advantage on serial programs, while if we measured parallel programs, we would be able to compare the two implementations in a broader context of their 'real' performance.

We have chosen to measure serial programs in spite of those objections because it is relatively easy to measure and understand such programs, and because of time constraints. Figure 5.1 shows the results of our measurements for the static data flow and combinator

Static Data Flow Program	Total Number of Node Firings	Time in Seconds	Node Firings Per Second
`Iter1`	16	≈ 2	≈ 8
`Iter5`	41	≈ 5	≈ 8
`Iter50`	356	≈ 34	≈ 10
`Cons1`	9	≤ 1	≥ 9
`Cons2`	38	≈ 5	≈ 7

Combinator Program	Total Number of Node Firings	Time in Seconds	Node Firings Per Second
`(Dfib 1)`	62	11.3	5.5
`(Dfib 2)`	130	≈ 22	≈ 5.9
`(Sfib 1)`	142	≈ 26	≈ 5.4

See the text for a description of the programs.

Figure 5.1: Performance Measurements on a few programs.

reduction simulators.

We measured two classes of programs for the static data flow simulator: The first class is an iteration: The program named `iter5` was five iterations of a simple addition, while the program named `iter1` was one iteration of the same program, and so on. The second class was a pair of programs which did some consing, and they appear in Figure 5.2.

We measured both the doubly recursive, and the singly recursive implementation of the Fibonacci function on the combinator simulator. The doubly recursive Fibonacci program is named `DFib`, and the singly recursive Fibonacci program is named `SFib`. The doubly recursive program has a lot of parallelism in it.

Our results indicate that our simulators are pretty slow. The number of firings in a static data flow graph is not directly comparable to the number of reductions in a combinator simulator, since every firing in the static data flow graph does seems to do more work than a

```
(defun cons1 ()
  (let ((x (cons 1 2)))
    (+ (car x) (cdr x))))

(defun cons2 ()
  (let ((x (cons (cons 1 2) (cons 3 4))))
    (+ (+ (car (car x))
          (cdr (car x)))
       (+ (car (cdr x))
          (cdr (cdr x))))))
```

Figure 5.2: Static data flow programs to measure consing.

combinator reduction: Many of the reductions in a combinator program are just the routing of arguments to the body of a function, while most of the firings in a static data flow graph are more related to the actual work which we want done. These results also indicate that relatively few of the PE's which are in a graph are actually active at any time.

We found that it was very easy to run out of processors for programs with much parallelism. For example, on a 512 processor connection machine, we ran out of processors while computing `(DFIB 5)`.

The preliminary measurements show that the static data flow interpreter is much faster than the combinator reduction interpreter. On the other hand, the static data flow paradigm is not as powerful as the combinator paradigm, because for example, it is not possible to write higher order functions or recursive functions in a static data flow language. We believe that the static data flow interpreter is inherently faster than combinator interpreters, because to do very simple operations with combinators can require a lot of work.

Supercombinators [Hug82] are one proposed solution which reduces the amount of work done for the simple operations, but the connection machine is not very well suited to simulating supercombinators: In general, the more rules there are to interpret, the longer the interpreter cycle takes to give all the nodes a chance to fire, and supercombinator compilers tend to produce a lot of rules.

5.2 Qualitative Results

We are able to run applicative programs, which allows us to gain experience with these programs. We are able to quickly test new ideas about the design of applicative architectures and other MIMD machines. These simulators also demonstrate that in the connection machine is a general purpose, efficient, scalable, parallel computer (as defined in Chapter 1), because it can run applicative programs efficiently.

We have found that the static data flow simulator runs a given program more quickly on our simulators than the combinator reduction simulator does. Thus, if a given program can be written in a static data flow language, such as VAL, it should be run on a static data flow machine.

One important lesson that we have learned is that the choice of message sending primitives can make a big difference on how easy it is to write programs. The use of `cm:get` and `cm:send-with-add` resulted in much simpler programs which were much easier to write, debug, and understand. In essence, these operations helped the connection machine look like a SIMD machine with a globally accessible memory (i.e. a PRAM) rather than a message passing SIMD machine.

We noticed that not very many PE's were really active at any given time. Presumably it would help the performance of these simulators to have more PE's active. Since we have mapped each primitive object in our programming model into a single PE (e.g. we mapped each node of a data flow graph, each cons cell, and each function application cell into one PE), there is not much we can do to increase the number of active PE's. It is possible that with an efficient implementation of virtual processors (see Section 2.2.6) we could use a higher percentage of the real PE's. The main architectural obstacle preventing efficient support of virtual processors is that each physical PE in the connection machine must address the same memory location at the same time. This means that if a virtual PE with its memory stored at location 0 in physical PE number 5, and another virtual PE with its memory stored at location 128 in physical PE number 10 both want to run at the same time, that the microcontroller

will have to issue the instructions twice: once for each virtual processor *bank*. It would help if each PE could access a different memory location, because then different banks could run on the different processors at the same time.

Clearly, it would help if the connection machine were a MIMD machine, since each PE could simulate a different kind of node more efficiently, and it would be feasible to implement supercombinators on the connection machine. Of course, it is not clear that if the connection machine were somehow changed into a MIMD machine that it would be anything like what we currently think of as a connection machine.

5.3 Future Work

Our simulators for static data flow and combinator reduction are running on the connection machine, and have largely met our goals. Our implementation for the VIM style dynamic data flow simulation has not been completed.

We are still interested in comparing the performance of VIM style dynamic data flow to combinators. It makes no sense to compare VIM style dynamic data flow with static data flow because static data flow is not as powerful as dynamic data flow, and anything that can be written in a static data flow language should run at least as quickly on a static data flow interpreter as on a VIM style dynamic data flow interpreter. For about the same reasons it makes little sense to compare combinator reduction with static data flow. However, it does make sense to compare combinator reduction to VIM style dynamic data flow.

The remaining problems that we have not answered to our satisfaction are

- What is the best way to limit the consumption of resources while allowing the parallelism of programs to be exploited.
- What is the tradeoff between a few simple combinators (which may require many reductions in order to do a given amount of work), and many powerful combinators (which require fewer reductions, but the interpreter cycle becomes longer)? Where in this

spectrum is the optimum?

- What is the best way to get the most computation per dollar spent on hardware. In SIMD architectures, this translates to a problem of using a high percentage of the PE's, and while our StarTalk compiler addresses this issue by performing optimizations, we have yet to see conclusive results on the design tradeoffs between various SIMD architectures such as the connection machine and MIMD architectures such as the tagged token data flow system[ACI*83].

- How bad is the code generated by the StarTalk compiler? Qualitatively, the code looks bad. Presumably, cleaning up the code would help improve the performance of our simulators. It is also possible that special purpose microcode would help the performance of our simulators. Further exploration of these efficiency issues is needed.

- What changes to the connection machine architecture would help applicative languages run quickly on the connection machine? Would a connection machine with such changes still be a *connection machine*?

Bibliography

[ACI*83] Arvind, D. E. Culler, R. A. Iannucci, V. Kathail, K. Pingali, and R. E. Thomas. The tagged token data flow architecture. 1983. Preliminary version for distribution in subject 6.843 at the Massachusetts Institute of Technology.

[AD79] William B. Ackerman and Jack B. Dennis. *VAL – A Value Oriented Algorithmic Language: Preliminary Reference Manual.* Technical Report MIT/LCS/TR-218, Massachusetts Institute of Technology, Laboratory for Computer Science, June 1979.

[ADI83] Arvind, Michael L. Dertouzos, and Robert A. Iannucci. *A Multiprocessor Emulation Facility.* Technical Report MIT/LCS/TR-302, Massachusetts Institute of Technology, Laboratory for Computer Science, October 1983.

[AI85] Arvind and R. A. Iannucci. *Two Fundamental Issues in Multiprocessing: The Dataflow Solution.* Laboratory for Computer Science, Computation Structures Group Memo 226-3, Massachusetts Institute of Technology, August 1985.

[Arv] Arvind. Personal Communication.

[Baw84] Alan Bawden. *A Programming Language for Massively Parallel Computers.* Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1984.

[BRR] Guy Blelloch, Abhiram Ranade, and John Rose. Personal communication. Guy Blelloch is currently a graduate student at M.I.T., Abhiram Ranade is currently a graduate student at Yale, and John Rose is currently employed at Thinking Machines Corporation, Cambridge, MA.

[Chr84] D. P. Christman. *Programming The Connection Machine.* Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1984.

[Den74] J. B. Dennis. First version of a data flow procedure language. In *Lecture Notes in Computer Science*, pages 362–376, Springer-Verlag, 1974.

[Den80] J. B. Dennis. Data flow supercomputers. *IEEE Computer*, 48–56, November 1980.

[DGK86] Susan Dickey, Allan Gottlieb, and Richard Kenner. Using vlsi to reduce serialization and memory traffic in shared memory parallel computers. In Charles E. Leiserson, editor, *Proceedings of the Fourth MIT Conference on Advanced Research in VLSI*, pages 299–316, April 7-9 1986.

[DSG85] Jack Dennis, Joseph Stoy, and Bhaskar Guharoy. Vim: an experimental multiuser system supporting functional programming. In *Proceedings of the International Workshop on High-Level Computer Architecture at Los Angeles California*, May 23-25 1985.

[Hil85] W. D. Hillis. *The Connection Machine.* The MIT Press, Cambridge, MA, 1985.

[Hug82] R. J. M. Hughes. Super-combinators. In *Proeedings of the 1982 Lisp and Functional Languages Conference*, 1982.

[Kus84] Bradley C. Kuszmaul. *Type Checking in VimVal.* Technical Report MIT/LCS/TR-332, Massachusetts Institute of Technology, Laboratory for Computer Science, May 1984.

[Kus85] Bradley C. Kuszmaul. Fast deterministic routing on hypercubes with small buffers. Term paper for Ron Rivest's graduate class on Advanced Algorithms. The paper is available from the author at the MIT Laboratory for Computer Science, Cambridge, MA 02142, December 1985.

[Tra84] K. R. Traub. An abstract architecture for parallel graph reduction. B.S. Thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 1984.

[Tur79] D. A. Turner. A new implementation technique for applicative languages. *Software-Practice and Experience*, 9:32–49, 1979.

Biographic note: Bradley "Bradley Bear" C. Kuszmaul was born in Iowa City, Iowa, where at the age of one year, he escaped from the watchful eyes of his parents, went outside, and got lost in the snow. His parents found him and took him to Kansas City, where he lived until migrating to Boston to attend M.I.T. Bradley Bear double majored in math and computer science at M.I.T., graduating in 1984 he won the runner up for best undergraduate thesis in the department of computer science (*Type Checking in* VIMVAL), which is probably the only reason that M.I.T. let him enter graduate school. He has subsequently written two papers: *The Suitability of the Connection Machine for Simulating Applicative Architectures*, and *Fast Deterministic Routing on Hypercubes with Small Buffers*. Bradley Bear received his nickname from mathematician Kimberly Kay Lewis, to whom he is now married.